Wandering in the Labyrinth of Thinking
– a cognitive architecture combining reinforcement learning and deep learning

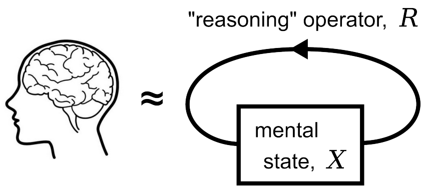
甄景贤 (King-Yin Yan) and Juan Carlos Kuri Pinto

General.Intelligence@Gmail.com

Abstract. This is a draft.

1 Main idea

The main idea is to regard "thinking" as a **dynamical system** operating on **mental states**:



(1)

For example, a mental state could be the following set of propositions:

- I am in my room, writing a paper for AGI-16.
- I am in the midst of writing the first sentence, "The main idea is..."
- I am about to write an infinitive phrase "to regard..."

Thinking is the process of **transitioning** from one mental state to another.

By representing a cognitive state as a vector $\vec{x} \in X$ where X is the cognitive state-space, the reasoning operator R as an **iterative map** $X \to X$, we would have at disposal all the tools available in vector space such as:

- numerical optimization (eg gradient descent)
- differential equations governing time evolution
- dynamical systems theory, control theory, dynamic programming, reinforcement learning
- neural networks and deep learning ... etc.

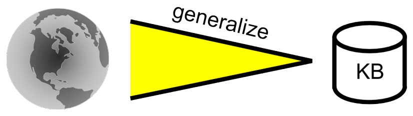
Related work: Google's **PageRank** is one of the earlier successful applications of vector-space and matrix techniques. The **Word2Vec** [3] algorithm that maps natural-language words to vectors is also spectacularly successful and influential; it demonstrated the potential advantages of vector representations. As for reinforcement learning, Q-learning (a form of RL) has been combined with deep learning to successfully play Atari games [**?**]; this model is exactly the same as our model, except that we are trying to refine the internal structure of the learner.

2 Logic-based AI (LBAI)

Main ideas:

- We would not directly implment logic-based AI, but it serves as a *backdrop* for understanding what are the problems of general AI.
- In this paper we would jump back and forth between the logic-based view and the dynamical state-space view. Knowledge of LBAI is essential to understanding ideas in this paper.

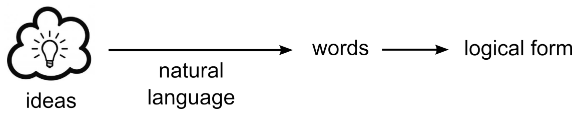
LBAI can be viewed as the compression of a world model into a knowledge-base (KB) of logic formulas:



(2)

The world model is *generated* combinatorially from the set of logic formulas, vaguely reminiscent of a "basis" in vector space. The generative process in logic is much more complicated, contributing to its high *compressive* ability on the one hand, and the *complexity* of learning such formulas on the other hand.

In LBAI the knowledge representation structure is built (*fixed*) from the bottom up:



(3)

but is it valid (or profitable) to assume that our mental representations are *isomorphic* to such logical structures? Or drastically different?

Humans are good at designing symbolic structures, but we don't know how to design *neural* representations which are more or less opaque to us. Perhaps we could use a neural network acting recurrently on the state vector to **induce** an internal representation of mental space. "*Induced by what*," you ask? By the very structure of the neural network itself. In other words, forcing a neural network to *approximate* the ideal operator R^*.

From an abstract point of view, we require:

- R be an endomorphism: $X \to X$
- R has a learning algorithm: $R \overset{A}{\longmapsto} R^*$

R would contain all the knowledge of the KB, so we expect it to be "large" (eg. having a huge number of parameters). We also desire R to possess a **hierarchical** structure because hierarchies are computationally very efficient. A multi-layer perceptron (MLP) seems to be a good candidate, as it is just a bunch of numbers (weight matrices W) interleaved by non-linear activation functions:

$$R(\boldsymbol{x}) = \circledS(W_1 \circledS(W_2 ... \circledS(W_L \boldsymbol{x}))) \tag{4}$$

where L is the number of layers. MLPs would be our starting point to explore more design options.

In 1991 Siegelmann and Sontag [2] proved that recurrent neural networks (RNNs) can emulate any Turing machine. In 1993 James Lo [1] proved that RNNs can universally approximate any non-linear dynamical system.

The idea of R as an operator acting on the state is inspired by the "consequence operator" in logic, usually denoted as Cn:

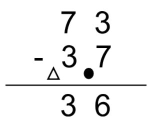
$$\text{Cn}(\Gamma) = \{ \text{ set of propositions that entails from } \Gamma \ \} \tag{5}$$

but the function of R can be broader than logical entailment. We could use R to perform the following functions which are central to LBAI:

- **deduction** – forward- and backward-chaining
- **abduction** – finding explanations
- **inductive learning**

Example 1: primary-school arithmetic

A recurrent neural network is a *much more powerful* learning machine than a feed-forward network, even if they look the same superficially.



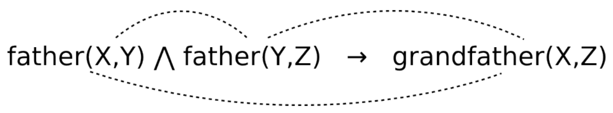
As an example, consider the way we perform 2-digit subtraction in primary school. This is done in two steps, and we put a dot on paper to mark "carry-over".

The use of the paper is analogous to the "tape" in a Turing machine – the ability to use short-term memory allows us to perform much more complex mental tasks.

We did a simple experiment to train a neural network to perform primary-school subtraction. The operator is learned easily if we train the two steps *separately*. The challenge is to find an algorithm that can learn **multi-step** operations by itself.

Example 2: variable binding in predicate logic

The following formula in predicate logic defines the "grandfather" relation:



(6)

We did a simple experiment to train a neural network to perform primary-school subtraction. The operator is learned easily if we train the two steps *separately*. The challenge is to find an algorithm that can learn **multi-step** operations by itself.

In LBAI, logic possesses additional structure:

– **truth values** (eg. P(rain tomorrow) = 0.7)
– **propositional structure** (eg. conjunction: $A \wedge B$)
– **sub-propositional structure** (eg. predication: loves(john, mary))
– **subsumption structure** (eg. dog $\subseteq$ animal)

These structures can be "transplanted" to the vector space X via:

– **truth values:** an extra dimension conveying the "strength" of states
– **propositional structure:** eg. conjunction as vector addition,

$$A \wedge B \quad \Leftrightarrow \quad \boldsymbol{x}_A + \boldsymbol{x}_B + \ldots \tag{7}$$

but we may have to avoid linear dependencies ("clashing") such as:

$$\boldsymbol{x}_3 = a_1 \boldsymbol{x}_1 + a_2 \boldsymbol{x}_2 \tag{8}$$

This would force the vector space dimension to become very high.

- **sub-propositional structure:** eg. tensor products as composition of concept atoms:

$$\text{loves(john, pete)} \quad \Leftrightarrow \quad \overrightarrow{john} \otimes \overrightarrow{love} \otimes \overrightarrow{pete} \tag{9}$$

- **subsumption structure:** eg. define the positive cone C such that

$$\text{animal} \supseteq \text{dog} \quad \Leftrightarrow \quad \overrightarrow{animal} - \overrightarrow{dog} \in C \tag{10}$$

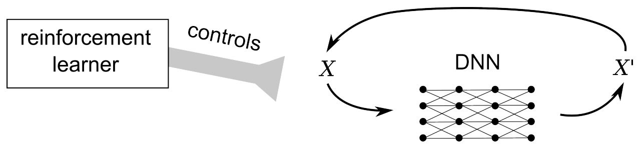
But the more logical structure we add to X, the more it will resemble logic, and this whole exercise becomes pointless. Remember our original goal is to try something different from logic, by *relaxing* what defines a logical structure. So we would selectively add features to X.

Andrew: This is where I want to formalize a "logical system". Particularly, the state X has internal structure that I have ignored so far: X should be a **set** of propositions. During deduction, we need to **select** a few propositions from X and try to **match** them with existing logic rules (this is the job of the famous **unifcation** algorithm in logical AI systems). The selection is part of the control variable u (see below). We need to decompose the vector X into some analogue of "propositions", but I don't know how to do it yet. Perhaps elucidating the algebraic form of the logic system will help us design the "vectorization" scheme.

3 Control theory / reinforcement learning

Basic ideas:

- Intelligence is decomposed into **thinking** and **learning**.
- **Thinking** is governed by control theory (finding the best trajectory in "thoughts space") under the contraints of correct reasoning, ie, knowledge.
- The iterative "thinking operator" is implemented as a deep-**learning** neural network (DNN). This DNN *contrains* the dynamics of thinking, and it represents the totality of *knowledge* in the system.



(11)

Our **metaphor** here is that of reinforcement learning controlling an autonomous agent to navigate the maze of "thoughts space":

(12)

3.1 What is control theory?

A **dynamical system** can be defined by:

$$\text{discrete time:} \quad \boldsymbol{x}_{t+1} = \boldsymbol{F}(\boldsymbol{x}_t) \tag{13}$$

$$\text{continuous time:} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \tag{14}$$

($\boldsymbol{F}$ is implemented as the deep learning network in our approach.)

A **control system** can be defined as (sometimes I mix with continuous-time notation for the sake of simplicity):

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \tag{15}$$

where $\boldsymbol{u}(t)$ is the **control vector**. The goal of control theory is to find the optimal $\boldsymbol{u}^*(t)$ function, such that the system moves from the initial state $\boldsymbol{x}_0$ to the terminal state $\boldsymbol{x}_\perp$.

3.2 What is reinforcement learning?

Reinforcement learning is synonymous with **dynamic programming**, which is also the main content of modern **control theory** with the state-space description.

The goal of **reinforcement learning** is to learn the **policy function**:

$$\pi : \boxed{\text{state}} \to \boxed{\text{action}} \tag{16}$$

when we are given the **state space**, **action space**, and **reward function**:

$$R : \boxed{\text{state}} \times \boxed{\text{action}} \to \mathbb{R} \tag{17}$$

The action a is the same notion as the control variable u in control theory.

The **Bellman equation** governs reinforcement learning just as in control theory:

$$\boxed{\text{optimal path}} = \text{choose max reward on current path segment} + \boxed{\text{the rest of optimal path}} \tag{18}$$

In math notation:

$$U_t^* = \max_u \{ \boxed{\text{reward(u, t)}} + U_{t-1}^* \} \tag{19}$$

where U is the "long-term value" or **utility** of a path.

Conceptually, U is the **integration** of instantaneous rewards over time:

$$\boxed{\text{utility / value U}} = \int \boxed{\text{reward R}}\, dt \tag{20}$$

3.3 Connection with Hamiltonian mechanics

An interesting insight from control theory is that our system is a Hamiltonian dynamical system in a broad sense.

Hamilton's **principle of least action** says that the trajectories of dynamical systems occuring in nature always choose to have their action S taking **stationary values** when compared to neighboring paths. The action is the time integral of the Lagrangian L:

$$\boxed{\text{Action}}\ S = \int L\, dt \tag{21}$$

From this we see that the Lagrangian corresponds to the instantaneous "rewards" of our system. It is perhaps not a coincidence that the Lagrangian has units of **energy**, in accordance with the folk psychology notion of "positive energy" when we talk about desirable things.

The **Hamiltonian** H arises when we consider a typical control theory problem; The system is defined via:

$$\text{state equation:} \quad \dot{\boldsymbol{x}}(t) = \boldsymbol{f}[\boldsymbol{x}(t), \boldsymbol{u}(t), t] \tag{22}$$

$$\text{boundary condition:} \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0\, ,\ \boldsymbol{x}(t_\perp) = \boldsymbol{x}_\perp \tag{23}$$

$$\text{objective function:} \quad J = \int_{t_0}^{t_\perp} L[\boldsymbol{x}(t), \boldsymbol{u}(t), t] dt \tag{24}$$

The goal is to find the optimal control $\boldsymbol{u}^*(t)$.

Now apply the technique of **Lagrange multipliers** for finding the maximum of a function, this leads to the new objective function:

$$U = \int_{t_0}^{t_\perp} \{L + \boldsymbol{\lambda}^T(t)\, [f(\boldsymbol{x}, \boldsymbol{u}, t) - \dot{\boldsymbol{x}}]\} dt \tag{25}$$

So we can introduce a new scalar function H, ie the Hamiltonian:

$$H(\boldsymbol{x}, \boldsymbol{u}, t) = L(\boldsymbol{x}, \boldsymbol{u}, t) + \boldsymbol{\lambda}^T(t) f(\boldsymbol{x}, \boldsymbol{u}, t) \tag{26}$$

Physically, the unit of $\boldsymbol{f}$ is velocity, while the unit of L is energy, therefore $\boldsymbol{\lambda}$ should have the unit of **momentum**. This is the reason why the phase space is made up of the diad of (position, momentum).

In its most general form we have the **Hamilton-Jacobi-Bellman equation**:

$$\boxed{\text{Hamilton-Jacobi-Bellman}} \quad 0 = \frac{\partial U^*}{\partial t} + \min_u H \tag{27}$$

To digress a bit, this equation is also analogous to the **Schrödinger equation** in quantum mechanics:

$$i\hbar \frac{\partial}{\partial t} \Psi(x, t) = \left[V(x, t) + \frac{-\hbar^2}{2\mu} \nabla^2 \right] \Psi(x, t). \tag{28}$$

where Ψ is analogous to our U (perhaps Ψ is something that nature wants to optimize?)

All these "physical" ideas follow automatically from our definition of **rewards**, without the need to introduce them artificially. But these ideas seem not immediately useful to our project, unless we are to explore **continuous-time** models.

Andrew: as you can see, the control theory part is essentially separated from the "logic" aspects.

From the viewpoint of reinforcement learning, we aim to learn the **policy** function:

$$\text{policy}: \quad \text{state} \xmapsto{\text{action}} \text{state'} \tag{29}$$

Where K can be regarded as the **mental state**, and thus an **action** in RL turns K into K'.

In our system, there are 2 pathways that act on K, via RNN and RL respectively:

$$\begin{tikzcd} & K'_1 \arrow[dd, dash, dashed, "\approx"] \\ K \arrow[ur, "\text{RL}"] \arrow[dr, "\text{RNN}"'] & \\ & K'_2 \end{tikzcd} \tag{30}$$

In RL, the action a acts on K, whereas in RNN, R acts on K.

Note: RNN and RL are learning algorithms, and if they are both applied to the same problem, conflicts will necessarily arise, unless there is a way to combine them.

At state K, we estimate the Q-value $Q(K \overset{a}{\mapsto} K')$. The action that would be chosen at state K is $\arg\max_a Q(K \overset{a}{\mapsto} K')$. This could be used to train the RNN via $K \vdash_W \ldots^n K'$.

RL 在众多状态 K 之间游荡，学习 $Q(K \mapsto K')$。因为 RL 独有奖励讯息，我们必需用 RL 来教导 RNN 学习，反之不可。第一个问题是：RL 如何在 K 之间游荡？游荡是随机的，但也可以借助 RNN 的随机性、或在 RNN 自身的游荡中注入更多随机性、或者根本就是 RL 自己产生的随机性。接下来的问题是：RNN 如何用 Q 值来诱发学习？

RNN 的 “n-fold” 学习可以通过以下方式实现：

- stochastic forward-backward propagation
- genetic?
- 最有趣的是 Hebbian learning，因为它似乎特别适合这情况。

RNN 的本质是什么？它似乎是一个 recurrent hetero-associative memory。但其实它还需要将 input 作类似於 Word2vec 的 encoding。这个 encoding 将「相似」的思维状态 K 归到同类。利用空间中的相似度，RL 可以用一些连续函数来近似 Q 值（详细情况还有待分析）。

另一个问题是：虽然用函数的近似可以做到 generalization，但另一个方法是利用状态 K 中的空位作暂时储存。这两者似乎很不同。问题似乎在於：状态转换 $K \mapsto K'$ 是不是对应於逻辑中的**一条** rule？答案似乎是 yes。这个共识是很重要的。如果用 decision tree，需要的是向量空间中的相似度。

现在的关键是「状态变量」。因为它可以做到符号逻辑中靠变量的 generalization，这是前所未有的。这种 generalization 似乎不需要相似度，因为它是符号的！会不会在向量空间中的状态变量能够做到之前逻辑变量做不到的动作？不管怎样，用 RNN 学习这些变量的动作似乎是很难的，因为这些动作似乎不是对误差的梯度下降。除非这些动作本身也近似於其他动作，但那是怎样的近似？学习 multi-step logic 其实和以前的 forward / backward chaining 没有分别！唯一分别是命题的 representation 改变了，它未必像符号的 concatenation。所以问题仍然是 “n-fold” 学习法。

而且注意：RL 的 generalization 根本上不同於 rules 空间中的 generalization。前者是思维空间 K 中的一般化，后者也可以是 K 空间的一般化，但也可以是依赖「状态变量」的一般化。

一般来说，RL 和 RNN 的行动和学习，是可以互相独立的。

还有 heterarchical 的分类法。想用 decision tree 或什么，达到不同网络的分工。在组织知识这方面，深度网络有没有用？可以想像，在视觉识别中，在网络的最上层有很多 objects，而它们都可以还原到底层的 features。网络有更多层，可以识别的事物更抽象。但现在我们要的不是模式识别，而是 mapping。特别是抽象模式的 mapping。想要的是：大量的 rules，将不同的 K 映射到新的 K'。

还有一点要澄清的是：究竟每一个「思元素」在向量空间中是不是一点？如果有了这个「思元素 = 点」假设，则每次 iteration 应该会删除一个思元素，而用另一个（全新的）思元素取代之。这样，$K \mapsto K'$ mapping 就有了更确定的结构。这样的 setup 已经很接近 logic 系统，但其学习算法仍然很有 combinatorial 的 "feel"。（因为只有当两个 rules 串连之后，才能达到某个结论，而这个串连有没有中间的 continuous 状态？）这种串连通常是怎样找到的？

现在有一转机：如果「思元素 = 点」，则「状态变量」的形成似乎会很普遍，而我们可以集中研究如何学习 single-step rules。RL 的 rewards 可以指导学习，但这些「终极 rewards」对学习的细节没有指导作用。我们似乎可以用「时间延迟」来达到「状态变量」的效果，这个做法无形中增加了使用状态变量的机会。

现在总结一下仍然有待回答的问题：

- RL 的 generalization 如何做？
- iterative thinking map 如何 learn？
-

Hebbian 的情况是：有某一 I/O pattern；我想 strengthen 这 pattern。

Assuming the learning is correct, K'_1 and K'_2 should be roughly the same — but this ignored the possibility that one path may take multiple steps to converge with the other path. [1]

Now I stipulate that R be more "refined", that is to say, applying D^n times may be equivalent to applying a once:

$$\begin{tikzcd} & K'_1 \arrow[dd, dashed, no head, "\approx"] \\ K \arrow[ur, "a"] \arrow[dr, "D^n"'] & \\ & K'_2 \end{tikzcd} \tag{31}$$

[1] This situation has been encountered in term rewriting systems (TRS): If in a TRS any 2 different rewriting paths always converge to the same result, it is said to have the **Church-Rosser property**. For example the λ-calculus invented by Church has this property.

Using a different notation, a is the **restriction** or **section** of D^n at point K: $a = D^n|_K$.

Now the question is, do the RNN and RL paths have any *essential* difference?

- Their internal **representations** are different:
  — RNN is a multi-layer neural network
  — RL's representation is $Q(\text{state}, \text{action})$, usually stored as a *look-up table*, although Q could be approximated by a neural network as well.
- RL learns through **rewards**, RNN learns from **errors**. Thus RL has broader applicability, because not all questions have "correct answers" that could be measured by errors. In RL we just need to praise Genifer whenever she displays good behavior.
- The internal cognitive state K exists because of RNN: it is simply the vector input and output of the RNN. Without this K, RL would be clueless as to what are its internal states. It can be said that the RNN provides a *machinery* for RL to control.

From the perspective of reinforcement learning, we could reward some results of multi-step inference:

$$K_0 \xmapsto{a} K_\vdash \quad \updownarrow\bigstar \tag{32}$$

$\updownarrow\bigstar$ means "to give positive or negative rewards". We want to learn a which is the action to be taken at state K. The learning algorithm is based on the famous **Bellman optimality condition** (see next section).

Perhaps we should use RL to *guide* the learning in RNN, as RNN is more fine-grained....

To combine the 2 learning approaches, we could use the technique of **interleaving**: for each step apply RL once, apply RNN n times.

The learning in RNN may also involve **neurogenesis** (adding new neurons and connections), but I have not considered this aspect yet.

There are 4 learning modes:

- learning to listen/talk
- RL-based learning
- inductive learning

4 Misc points

- If sigmoid is replaced by polynomial, universal approximating property may be retained.
- Banach fixed point theorem does not apply because R in general need not be contractive. Question is whether R necessarily converges to fixed points and the answer is no.
- If reasoning operator R is continuous, the flow of the dynamical system is governed by an autonomous differential equation. Poincare-Bendixson only applies to dynamical systems on the plane, and is irrelevant to systems whose phase space has dimension ≥ 3, or to discrete dynamical systems.
- Time can be discrete or continuous.
- Goal is to find minimizer of error (ie, to approximate a function given some input-output data points). The (finite) set of local minima can be solved via setting $\frac{\partial R}{\partial W} = 0$. The number of local minima can be calculated as: ? McClelland paper.
- If operator is discontinuous, what advantages can be gained?

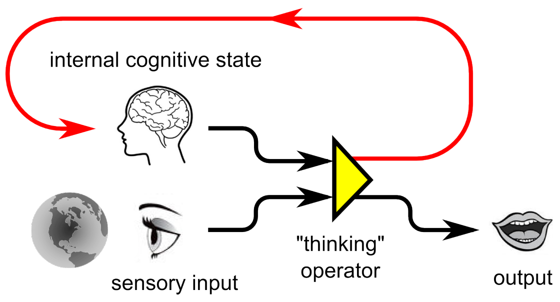
What I want to do now is to determine if R implemented as a deep network is sufficient to model human-level reasoning.

One principle seems to be that logical conclusions must not proliferate indefinitely. But we are not sure what kind of structural constraints this would impose on the vector space. Or whether we should impose such constraints manually.

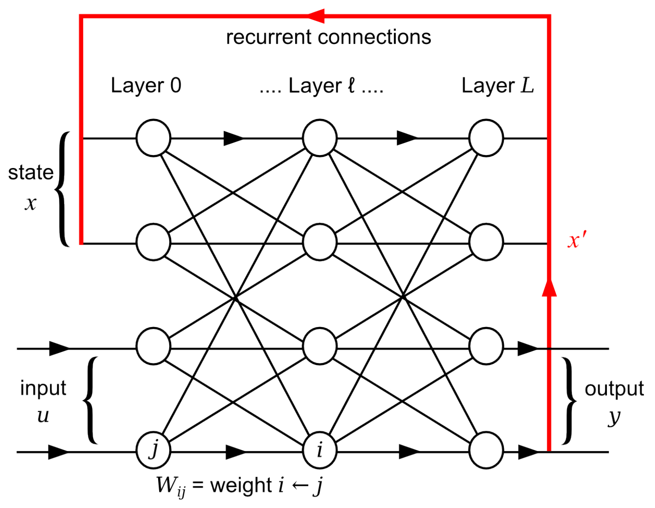
What other properties are desired for the implementation of R?

5 Architecture

First, cartoon version:

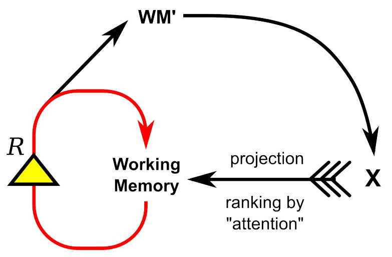


(33)



(34)

TO-DO: The state space X may be too large and we may need an **attention mechanism** to select some parts of X for processing by R. This is the notion of **working memory** in cognitive science.



(35)

6 Deep Recurrent Learning

The learning algorithm for R is central to our system. R learns to recognize input-output pairs $(\vec{x}_0, \vec{x}^*)$. What makes it special is that R is allowed to iterate a *flexible* number of times before outputting an answer. In feed-forward learning we simply learn single-pass recognition, whereas in common recurrent learning we train against a *fixed* time sequence. Here, the time delay between input and output is allowed to stretch arbitrarily.

Suppose the recurrent network R iterates n times:

$$\vec{x}_{t+1} = \overbrace{R \circ R \circ ...}^{n}(\vec{x}) \tag{36}$$

As $n \to \infty$, we get the continuous-time version (a differential equation):

$$\frac{d\vec{x}(t)}{dt} = \mathfrak{R}(\vec{x}(t)) \tag{37}$$

We could run the network R for a long enough time T such that it is highly likely to reach an equilibrium point. Then:

$$\vec{x}_T = \int_0^T \mathfrak{R}(\vec{x}(t))dt \tag{38}$$

and the error:

$$\mathscr{E} = \vec{x}^* - \vec{x}_T \tag{39}$$

where $\vec{x}^*$ is the target value which is independent of time.

$$\begin{aligned}\frac{\partial \mathscr{E}}{\partial \vec{W}} &= -\frac{\partial}{\partial \vec{W}} \int_0^T \mathfrak{R}(\vec{x}(t))dt \\ &= -\frac{\partial}{\partial \vec{W}} \int_0^T \circledS(W_1 \circledS(W_2 ... \circledS(W_L \vec{x}(t)))dt \end{aligned} \tag{40}$$

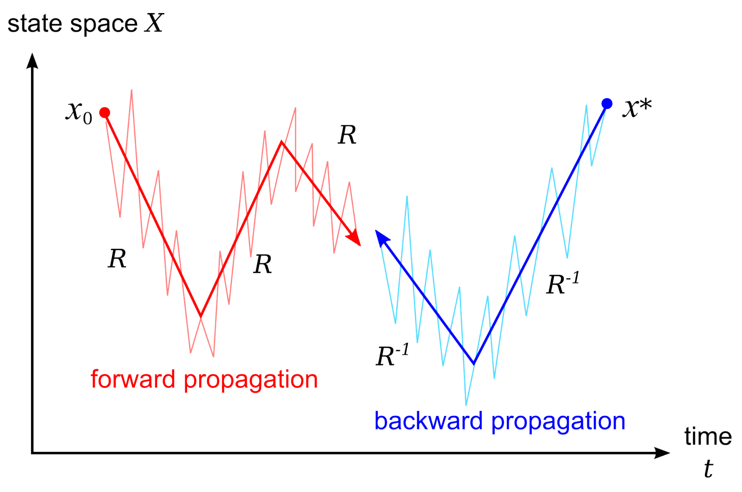
When there are many layers or if the recurrence is too long, back-prop learning becomes ineffective due to the **vanishing gradient** problem. One solution is to use the **rectifier** activation function:

$$\oslash(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{41}$$

Since its derivative is piecewise constant, it does not suffer from the vanishing gradient problem.

6.1 Forward-backward Algorithm

This is inspired by forward- and backward-chaining in LBAI. We propagate the state vector from both the initial state $\vec{x}_0$ as well as the final state $\vec{x}^*$. This bi-directional propagation is added with noise and repeated many times, thus implementing a **stochastic local search**:



(42)

When the forward and backward states get close enough, a successful path is found, and we record the gap and the noises along the path, and use them to train R so that this new path would be recognized.

Acknowledgements

In a forum discussion with Ben Goertzel dated 25 June 2014 on the AGI mailing-list: (agi@listbox.com), YKY asked: Why bother with neural networks, which typically require many neurons to encode data, when logic-based AI can represent a proposition with just a few symbols? Ben's insight is that neural networks are capable of learning its own representations, and their learning algorithms are relatively speaking much faster. We have been working on "neo-classical" logic-based AI for a long time, and began to realize that inductive learning in logic (based on combinatorial search in a symbolic space) is perhaps *the bottleneck* in the entire logic-based paradigm. So we try to look for alternatives that might enable learning to be faster, though we would still emphasize that logic-based AI remains a viable approach to AGI.

References

1. Lo. Dynamical system identification by recurrent multilayer perceptrons. *Proceedings of the 1993 World Congress on Neural Networks*, 1993.
2. Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller. Playing atari with deep reinforcement learning. *arXiv:1312.5602 [cs.LG]*, 2013.
3. Siegelmann and Sontag. Turing computability with neural nets. *Applied Mathematics Letters, vol 4, p77-80*, 1991.
4. Weston, Chopra, and Bordes. Memory networks. *ICLR (also arXiv)*, 2015.